Improving MCS Enumeration via Caching

Previti, Alessandro

Springer International Publishing AG
2017

Previti , A , Mencía , C , Järvisalo , M & Marques-Silva , J 2017 , Improving MCS Enumeration via Caching . in S Gaspers & T Walsh (eds) , Theory and Applications of þÿ S a t i s f i a b i l i t y T e s t i n g S A T 2 0 1 7 : 2 0 t h I n t e r n a t i o n a l C o n f e r e n c e M e l b þÿ A u g u s t 2 8 S e p t e m b e r 1 , 2 0 1 7 P r o c e e d i n g s . L e c t u r e N o t e s i n C o m p u t 10491 , Springer International Publishing AG , Cham , pp. 184-194 , International Conference on Theory and Applications of Satisfiability Testing , Melbourne , Australia , 28/08/2017 . https://doi.org/10.1007/978-3-319-66263-3_12

http://hdl.handle.net/10138/309055
https://doi.org/10.1007/978-3-319-66263-3_12

acceptedVersion

Improving MCS Enumeration via Caching*

Alessandro Previti[1], Carlos Mencía[2], Matti Järvisalo[1], and Joao Marques-Silva[3]

[1] HIIT, Department of Computer Science, University of Helsinki, Helsinki, Finland
[2] Department of Computer Science, University of Oviedo, Gijón, Spain
[3] LASIGE, Faculty of Science, University of Lisbon, Lisbon, Portugal

Abstract. Enumeration of minimal correction sets (MCSes) of conjunctive normal form formulas is a central and highly intractable problem in infeasibility analysis of constraint systems. Often complete enumeration of MCSes is impossible due to both high computational cost and worst-case exponential number of MCSes. In such cases partial enumeration is sought for, finding applications in various domains, including axiom pinpointing in description logics among others. In this work we propose caching as a means of further improving the practical efficiency of current MCS enumeration approaches, and show the potential of caching via an empirical evaluation.

1 Introduction

Minimal correction sets (MCSes) of an over-constrained system are subset-minimal sets of constraints whose removal restores the consistency of the system [6]. In terms of unsatisfiable conjunctive normal form (CNF) propositional formulas, the focus of this work, MCSes are hence minimal sets of clauses such that, once removed, the rest of the formula is satisfiable. Due to the generality of the notion, MCSes find applications in various domains where understanding infeasibility is a central problem, ranging from minimal model computation and model-based diagnosis to interactive constraint satisfaction and configuration [17], as well as ontology debugging and axiom pinpointing in description logics [1].

On a fundamental level, MCSes are closely related to other fundamental notions in infeasibility analysis. These include maximal satisfiable subsets (MSSes), which represent the complement notion of MCSes (sometimes referred to as co-MSSes [11]), and minimally unsatisfiable subsets (MUSes), with the well-known minimal hitting set duality providing a tight connection between MCSes and MUSes [4, 6, 26]. Furthermore, MCSes are strongly related to maximum satisfiability (MaxSAT), the clauses satisfied in an optimal MaxSAT solution being the residual formula after removing a smallest (minimum-weight) MCS over the soft clauses. Not surprisingly, MCS extraction surpasses in terms of computational complexity the task of satisfiability checking, deciding whether a given subset of clauses of a CNF formula is an MCS being DP-complete [7].

* A.P. and M.J. were supported by Academy of Finland (grants 251170 COIN, 276412, and 284591) and the Research Funds of the University of Helsinki. C.M. was supported by grant TIN2016-79190-R. J.M.S. was supported by FCT funding of LASIGE Research Unit, ref. UID/CEC/00408/2013.

Despite this, and on the other hand motivated by the various applications and fundamental connections, several algorithms for extracting an MCS of a given CNF formula have been recently proposed [3,11,17–20,22,24], iteratively using Boolean satisfiability (SAT) solvers as the natural choice for the underlying practical NP oracle.

In this work we focus on the computationally more challenging task of MCS enumeration. Complete enumeration of MCSes is often impossible due to both high computational cost and the worst-case exponential number of MCSes. In such cases partial enumeration is sought for, which finds many application domains, including axiom pinpointing in description logics [1] among others.

Instead of proposing a new algorithm for MCS enumeration, we propose the use of *caching* as a means of further improving the scalability of current state-of-the-art MCS enumeration algorithms. Caching (or memoization) is of course a well-known general concept, and has been successfully applied in speeding up procedures for other central problems related to satisfiability. A prime example is the use of subformula caching in the context of the #P-complete model counting problem [2, 5, 12, 13, 27, 30]. Similarly, clause learning in CDCL SAT solvers [23, 28] can be viewed as a caching mechanism where learned clauses summarize and prevent previously identified conflicts.

In more detail, we propose caching unsatisfiable cores met during search within SAT-based MCS enumeration algorithms. Putting this idea into practice, we show that core caching has clear potential in scaling up MCS enumeration, especially for those instances whose extraction of a single MCS is not trivial. In terms of related work, to the best of our knowledge the use of caching to scale up MCS enumeration has not been previously proposed or studied. Partial MUS enumerators (e.g. [14, 31]) store MUSes and MCSes in order to exploit hitting set duality and enumerate both. In contrast, we use caching to avoid potentially hard calls to a SAT solver.

The rest of this paper is organized as follows. In Section 2 we overview necessary preliminaries and notation used throughout, and in Section 3 provide an overview of MCS extraction and enumeration algorithms. We propose caching as a means of improving MCS enumeration in Section 4, and, before conclusions, present empirical results on the effects of using this idea in practice in Section 5.

2 Preliminaries

We consider propositional formulas in conjunctive normal form (CNF). A CNF formula $\mathcal{F}$ over a set of Boolean variables $X = \{x_1, ..., x_n\}$ is a conjunction of clauses $(c_1 \wedge ... \wedge c_m)$. A clause c_i is a disjunction of literals $(l_{i,1} \vee ... \vee l_{i,k_i})$ and a literal l is either a variable x or its negation $\neg x$. We refer to the set of literals appearing in $\mathcal{F}$ as $L(\mathcal{F})$. CNF formulas can be alternatively represented as sets of clauses, and clauses as sets of literals. Unless explicitly specified, formulas and clauses are assumed to be represented as sets.

A truth assignment, or interpretation, is a mapping $\mu : X \rightarrow \{0, 1\}$. If each of the variables in X is assigned a truth value, μ is a *complete* assignment. Interpretations can be also seen as conjunctions or sets of literals. Truth valuations are lifted to clauses and formulas as follows: μ satisfies a clause c if it contains at least one of its literals, whereas μ falsifies c if it contains the complements of all its literals. Given a formula

$\mathcal{F}$, μ satisfies $\mathcal{F}$ (written $\mu \models \mathcal{F}$) if it satisfies all its clauses, in which case μ is a *model* of $\mathcal{F}$.

Given two formulas $\mathcal{F}$ and $\mathcal{G}$, $\mathcal{F}$ entails $\mathcal{G}$ (written $\mathcal{F} \models \mathcal{G}$) if and only if each model of $\mathcal{F}$ is also a model of $\mathcal{G}$. A formula $\mathcal{F}$ is satisfiable ($\mathcal{F} \not\models \bot$) if it has a model, and otherwise unsatisfiable ($\mathcal{F} \models \bot$). SAT is the NP-*complete* [8] decision problem of determining the satisfiability of a given propositional formula.

The following definitions give central notions of subsets of an unsatisfiable formula $\mathcal{F}$ in terms of (set-wise) minimal unsatisfiability and maximal satisfiability [15, 17].

Definition 1. *$\mathcal{M} \subseteq \mathcal{F}$ is a* minimally unsatisfiable subset *(MUS) of $\mathcal{F}$ if and only if $\mathcal{M}$ is unsatisfiable and $\forall c \in \mathcal{M}, \mathcal{M} \setminus \{c\}$ is satisfiable.*

Definition 2. *$\mathcal{C} \subseteq \mathcal{F}$ is a* minimal correction subset *(MCS) if and only if $\mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall c \in \mathcal{C}, \mathcal{F} \setminus (\mathcal{C} \setminus \{c\})$ is unsatisfiable.*

Definition 3. *$\mathcal{S} \subseteq \mathcal{F}$ is a* maximal satisfiable subset *(MSS) if and only if $\mathcal{S}$ is satisfiable and $\forall c \in \mathcal{F} \setminus \mathcal{S}, \mathcal{S} \cup \{c\}$ is unsatisfiable.*

Note that an MSS is the set-complement of an MCS. MUSes and MCSes are closely related by the well-known hitting set duality [4, 6, 26, 29]: Every MCS (MUS) is an irreducible hitting set of all MUSes (MCSes) of the formula. In the worst case, there can be an exponential number of MUSes and MCSes [15, 25]. Besides, MCSes are related to the maximum satisfiability (MaxSAT) problem, which consists in finding an assignment satisfying as many clauses as possible; a smallest MCS (i.e., largest MSS) is the set of clauses left unsatisfied by some optimal MaxSAT solution.

Given the practical significance of handling soft constraints [21], we consider that formulas may be partitioned into sets of hard and soft clauses, i.e., $\mathcal{F} = \mathcal{F}_H \cup \mathcal{F}_S$. Hard clauses must be satisfied, while soft clauses can be relaxed if necessary. Thus, an MCS will be a subset of $\mathcal{F}_S$.

The following simple proposition will be useful in the remainder of this paper.

Proposition 1. *Let $\mathcal{M}$ be an unsatisfiable formula. Then, for any $\mathcal{M}' \supseteq \mathcal{M}$ we have that also $\mathcal{M}'$ is unsatisfiable.*

3 MCS Extraction and Enumeration

In this section we overview the state-of-the-art MCS enumeration algorithms. These algorithms work on a formula $\mathcal{F} = \mathcal{F}_H \cup \mathcal{F}_S$ partitioned into hard and soft clauses, $\mathcal{F}_H$ and $\mathcal{F}_S$, respectively. The hard clauses are added as such to a SAT solver. Each soft clause c_i is extended, or reified, with a fresh selector (or assumption) variable s_i, i.e. soft clause c_i results in the clause $(\neg s_i \vee c_i)$, before adding them to the SAT solver. The use of selector variables is a standard technique used to add and remove clauses, enabling incremental SAT solving. Selector variables are set as assumptions at the beginning of each call to the SAT solver in order to activate (add) or deactivate (remove) a clause. In particular, if s_i is set to 1, then the associated clause is activated. If s_i is set to 0, then c_i is deactivated. The addition of the selector variables makes $\mathcal{F}$ satisfiable (provided that

Algorithm 1: Basic linear search

```
1 Function BLS (F)
2     M ← ∅
3     (S,U) ← InitialAssignment(F)
4     foreach c ∈ U do
5         ⟨st, μ, C⟩ ← SAT(S ∪ {c})
6         if st then S ← S ∪ {c}
7         else M ← M ∪ {c}
8     return M                                  // MCS of F
```

$\mathcal{F}_H$ is satisfiable). When all the selector variables s_i are set to 1, the result is the original formula $\mathcal{F}$. MCS algorithms use the selector variables as assumptions for selecting subsets of $\mathcal{F}_S$ over which to check satisfiability together with the hard clauses. We will refer to the subset of soft clauses of $\mathcal{F}_S$ identified by a set of selector variables together with the hard clauses as the *induced formula*. In presenting the algorithms, we will avoid referring explicitly to selector variables; rather, we identify a formula $\mathcal{F}$ with all selector variables of the soft clauses in $\mathcal{F}_S$ being set to 1.

State-of-the-art MCS extraction algorithms rely on making a sequence of calls to a SAT solver that is used as a witness NP oracle. The solver is queried a number of times on subformulas of the unsatisfiable input formula $\mathcal{F}$. A SAT solver call is represented on line 5 by $\langle st, \mu, C\rangle \leftarrow \texttt{SAT}(\mathcal{F})$, where st is a Boolean value indicating whether the formula is satisfiable or not. If the formula is satisfiable, the SAT solver returns a model μ. Otherwise it returns an unsatisfiable core C over the soft clauses.

We overview in more detail a simple example of such algorithms: the basic linear search (BLS) approach, depicted in Algorithm 1. This algorithm maintains a partition of $\mathcal{F}$ in two disjoint sets during the computation of an MCS. The set $\mathcal{S}$ represents a satisfiable subformula of $\mathcal{F}$, i.e., the MSS under construction. The set $\mathcal{U}$ is formed by the clauses that still need to be checked. The initial assignment used to split $\mathcal{F}$ is a model μ of $\mathcal{F}_H$. All the clauses in $\mathcal{F}$ satisfied by μ are put in $\mathcal{S}$, while the falsified clauses become part of $\mathcal{U}$. These operations are enclosed inside the function $\texttt{InitialAssignment}(\mathcal{F})$ on line 3. Then, iteratively until all the clauses in $\mathcal{U}$ have been checked, the algorithm picks a clause $c \in \mathcal{U}$ and checks the satisfiability of $\mathcal{S} \cup \{c\}$. If it is satisfiable, c is added to $\mathcal{S}$. Otherwise, c is known to belong to the MCS under construction and is added to $\mathcal{M}$. Upon termination, $\mathcal{S}$ represents an MSS and $\mathcal{M} = \mathcal{F} \setminus \mathcal{S}$ represents an MCS of $\mathcal{F}$.

In linear search, the number of SAT solver calls necessary is linear in terms of the number of soft clauses in the input formula. Different alternatives and optimization techniques have been proposed in recent years, leading to substantial improvements, including FastDiag [10], dichotomic search [25], clause D (CLD) [17], relaxation search [3], and the CMP algorithm [11]. In addition, algorithms such as the literal-based extractor (LBX) [20] represent the current state-of-the-art for extracting a single MCS. Recently, algorithms such as LOPZ, UCD and UBS, which also target the extraction of a single MCS, have been proposed [18], requiring a sublinear number of SAT solver calls on the number of clauses. Optimization techniques include exploiting satisfying assignments, backbone literals, and disjoint unsatisfiable cores [17], among

others, and are integrated into MCS extraction algorithms for improving efficiency, giving rise to, e.g., enhanced linear search (ELS) [17].

MCS enumeration relies on iteratively extracting an MCS $\mathcal{C} \in \mathcal{F}$ and blocking it by adding the hard clause $\bigvee_{l \in L(\mathcal{C})} l$ to $\mathcal{F}$. This way, no superset of $\mathcal{C}$ will subsequently be considered during the enumeration. The process continues until $\mathcal{F}_H$ becomes unsatisfiable, at which time all MCSes have been enumerated.

To the best of our knowledge, the current state-of-the-art in MCS enumeration is represented by the algorithms implemented in the tool mcsls [17], specifically, ELS and CLD. These algorithms have been shown to be complementary to each other [16].

4 Caching for MCS Enumeration

We will now introduce caching as a way to improve MCS enumeration. For some intuition, when a formula has a large number of MCSes, many of the MCSes tend to share many clauses. This suggests that similar satisfiability problems are solved in the computation of several MCSes. Our proposal aims at making use of this observation by storing information that could lead to avoiding potentially time-consuming calls to the SAT solver on $\mathcal{S} \cup \{c\}$.

The idea is to keep a global database which is updated and queried by the MCS extraction algorithm during the enumeration process. The only requirements for realizing the database are the two operations *store(C)* and *hasSubset($\mathcal{K}$)*, where C is an unsatisfiable core of $\mathcal{F}$ and $\mathcal{K} \subseteq \mathcal{F}$. The intent of the function *hasSubset($\mathcal{K}$)* is to check for a given subset $\mathcal{K}$ of $\mathcal{F}$ whether an unsatisfiable core C of $\mathcal{F}$ with $C \subseteq \mathcal{K}$ has already been extracted. If this is the case, we know by Proposition 1 that $\mathcal{K}$ is unsatisfiable. Naturally, as the cache database queries should avoid the cost of calling a SAT solver on the actual instance, the functions *store(C)* and *hasSubset($\mathcal{K}$)* need to be fast to compute.

Considering these requirements, as well as ease of implementing the cache and queries to the cache, in this work we implement the database by means of a SAT solver,

Algorithm 2: Basic linear search with caching

Function BLS-CACHING ($\mathcal{F}$)
  Global: $\mathcal{D}$
  $\mathcal{M} \leftarrow \emptyset$ // MCS under construction
  $(\mathcal{S}, \mathcal{U}) \leftarrow$ `InitialAssignment`$(\mathcal{F})$
  foreach $c \in \mathcal{U}$ **do**
    $\mathcal{A} \leftarrow \{s_i | c_i \in S \cup \{c\}\}$
    $\langle st, \mu, C \rangle \leftarrow$ `SAT`$(\mathcal{D} \cup \mathcal{A})$
    if *not* st **then**
      $\mathcal{M} \leftarrow \mathcal{M} \cup \{c\}$
      continue
    $\langle st, \mu, C \rangle \leftarrow$ `SAT`$(\mathcal{S} \cup \{c\})$
    if *not* st **then**
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\bigvee_{c_i \in C} \neg\sigma(c_i))\}$
      $\mathcal{M} \leftarrow \mathcal{M} \cup \{c\}$
    else $\mathcal{S} \leftarrow \mathcal{S} \cup \{c\}$
  return $\mathcal{M}$ // MCS of $\mathcal{F}$

storing a formula referred to as $\mathcal{D}$ formula in Algorithm 2. Variables of this formula are the selector variables of the original formula $\mathcal{F}$, while clauses represent unsatisfiable cores of $\mathcal{F}$. For an unsatisfiable core C of $\mathcal{F}$ the corresponding clause is given by $(\bigvee_{c_i \in C} \neg\sigma(c_i))$, where $\sigma(c_i)$ is a function which for a given clause c_i returns the associated selector variable s_i. As an example, suppose that $C = \{c_1, c_2, c_3\}$ is an unsatisfiable core of $\mathcal{F}$. The corresponding clause added to $\mathcal{D}$ is $(\neg s_1 \vee \neg s_2 \vee \neg s_3)$. Notice that in $\mathcal{D}$ all literals are pure, since no positive literal is part of any clause. The $\mathcal{D}$ formula is in fact *monotone*. Consequently, checking the satisfiability of the $\mathcal{D}$ formula under any assumptions can be done in polynomial time. From a theoretical point of view, this clearly shows an advantage compared to calling a SAT solver on the formula $\mathcal{F}$.

Algorithm 2 shows the BLS algorithm extended with caching. The algorithm is here presented for simplicity in terms of extracting a single (next) MCS. To avoid repetition, we assume that the formula $\mathcal{F}_H$ contains blocking clauses of all the previously computed MCSes. As a consequence, any initial assignment computed at line 3 is guaranteed to split the formula in two parts $\mathcal{S}$ and $\mathcal{U}$ such that for any MCS $\mathcal{M} \subseteq \mathcal{U}$, $\mathcal{M}$ is not an already computed MCS.

Proposition 2. *Let $\mathcal{G}$ be a formula and $\mathcal{A} = \{s_i | c_i \in \mathcal{G}\}$. If $\mathcal{D} \cup \mathcal{A} \models \bot$, then $\mathcal{G} \models \bot$.*

Proof. Recall that each clause in $\mathcal{D}$ represents an unsatisfiable core. For $\mathcal{D} \cup \mathcal{A} \models \bot$ to hold, there has to be a clause $c \in \mathcal{D}$ whose literals are all falsified. This can happen if and only if we have $c \subseteq \mathcal{A}'$, where $\mathcal{A}' = \{\neg s_i | s_i \in \mathcal{A}\}$. Since the formula induced by c is unsatisfiable and $c \subseteq \mathcal{A}'$, by Proposition 1 it follows that $\mathcal{G}$ is unsatisfiable. □

Proposition 2 is applied on line 6 of Algorithm 2. In case $\mathcal{D} \cup \mathcal{A}$ is unsatisfiable, the call to the SAT solver on line 10 becomes unnecessary and we can immediately add c to $\mathcal{M}$ and proceed with testing the next clause. Otherwise, we are forced to test the clause c on the original formula (line 10). If $\mathcal{S} \cup \{c\}$ is unsatisfiable, we add the unsatisfiable core C to the formula $\mathcal{D}$ and c to the MCS under construction $\mathcal{M}$. If instead the outcome returned by the call is satisfiable, we add the clause to $\mathcal{S}$, the MSS under construction. When all clauses have been tested, the MCS $\mathcal{M} = \mathcal{F} \setminus \mathcal{S}$ is returned.

Example 1. Assume that $\mathcal{F} = \{c_1, c_2, c_3, c_4, c_5\}$ is an unsatisfiable formula with $M_1 = \{c_1, c_2, c_3\}$ and $M_2 = \{c_1, c_4, c_5\}$ the only MUSes of $\mathcal{F}$. An example run of Algorithm 2 is shown in Table 1. First $\mathcal{D}$ is empty and a SAT call on the original formula is made to identify c_3 as part of the MCS under construction. The query returns UNSAT, c_3 is added to the MCS under construction, and the unsatisfiable core $\{c_1, c_2, c_3\}$ is added to $\mathcal{D}$. For testing $\mathcal{S} \cup \{c_5\} = \{c_1, c_4, c_2, c_5\}$, $\mathcal{D} \cup \mathcal{A}$ (represented in CNF as $(\neg s_1 \vee \neg s_2 \vee \neg s_3) \wedge s_1 \wedge s_4 \wedge s_2 \wedge s_5$) is satisfiable, so another SAT call on $\mathcal{F}$ is required. This adds the additional unsatisfiable core $\{c_1, c_4, c_5\}$ to $\mathcal{D}$ and c_5 to $\mathcal{M}_1$, which is now a complete MCS. Finally, when testing clauses c_3 and c_4 (for the next MCS), we have that in both cases $\mathcal{D} \cup \mathcal{A}$ is unsatisfiable and the two clauses are added to $\mathcal{M}_2$. This example shows that while two SAT solver calls are needed for determining the first MCS, for the second one it suffices to query the core database. ■

Table 1: Example execution of Algorithm 2.

$\mathcal{S} \cup \{c\}$	$\mathcal{D}$	Query	$\mathcal{M}_1$
$\{c_1, c_4, c_2\} \cup \{c_3\}$	$\emptyset$	$\mathcal{F}$: UNSAT	$\{c_3\}$
$\{c_1, c_4, c_2\} \cup \{c_5\}$	$\{c_1, c_2, c_3\}$	$\mathcal{F}$: UNSAT	$\{c_3, c_5\}$
$\mathcal{S} \cup \{c\}$	$\mathcal{D}$	Query	$\mathcal{M}_2$
$\{c_1, c_2, c_5\} \cup \{c_3\}$	$\{c_1, c_2, c_3\}, \{c_1, c_4, c_5\}$	$\mathcal{D} \cup \mathcal{A}$: UNSAT	$\{c_3\}$
$\{c_1, c_2, c_5\} \cup \{c_4\}$	$\{c_1, c_2, c_3\}, \{c_1, c_4, c_5\}$	$\mathcal{D} \cup \mathcal{A}$: UNSAT	$\{c_3, c_4\}$

5 Experimental Results

We implemented the proposed approach (Algorithm 2), mcscache-els, on top of the state-of-the-art MCS enumeration tool mcsls [17] in C++, extending the ELS algorithm to use a core database for caching, and using Minisat 2.2.0 [9] as a backend solver. We implemented two optimizations: we use (i) satisfying assignments obtained from satisfiable SAT solver calls to extend the set $\mathcal{S}$ with all clauses in $\mathcal{U}$ satisfied by the assignments, and (ii) disjoint unsatisfiable cores by computing a set of disjoint cores at the beginning of search, which can lead to avoiding some calls to the SAT solver during the computation of MCSes. The current implementation does not use the so-called backbone literals optimization [17] with the intuition that this would make the core database non-monotone and thereby queries to the cache potentially more time-consuming.

We compare mcscache-els with two state-of-the-art approaches: ELS, which is the basis of mcscache-els, and CLD [17]. Both ELS (mcsls-els) and CLD (mcsls-cld) are implemented in the tool mcsls [17]. All algorithms accept formulas with soft and hard clauses. As benchmarks, we used the 811 instances from [17] for which an MCS could be extracted. These instances were originally used for benchmarking algorithms for extracting only a single MCS. Note that, in terms of MCS enumeration, these are therefore hard instances, and we expect caching to be most beneficial on such hard instances. The experiments were run on a computing cluster running 64-bit Linux on 2-GHz processors using a per-instance memory limit of 8 GB and a time limit of 1800 seconds.

We compare the performance of the algorithms in terms of the number of MCSes enumerated within the per-instance time limit. A comparison of mcscache-els and mcsls-els is shown in Fig. 1. Using caching clearly and consistently improves performance: with only few exceptions, caching enables enumerating higher numbers of MCSes. Note that the only difference between mcscache-els and mcsls-els is that the first uses the caching approach proposed in this work. We also compare mcscache-els to mcsls-cld; in this comparison the base algorithms are different. As can be observed from Fig. 2, mcsls-cld exhibits better performance for instances on which a lower number of MCSes are enumerated. However, as the number of MCSes enumerated increases, the performance of mcsls-cld noticeably degrades compared to mcscache-els and mcscache-els starts to clearly dominate.

Finally, we consider more statistics on the effects of caching. First, we observed that on a significant number of the instances the number of cache misses (cache queries which do not find the core queried for in the database) was very low, i.e., the success rate in querying the cache was high, as $> 90\%$ of MCS clauses were often detected from

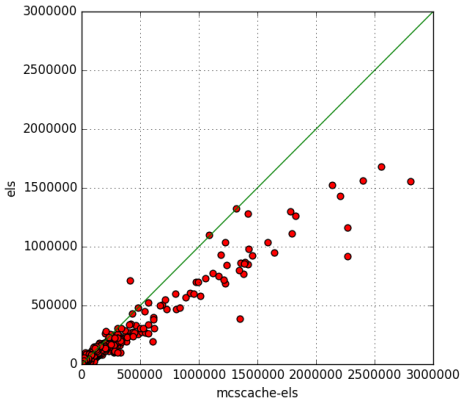

Fig. 1: mcscache-els vs mcsls-els

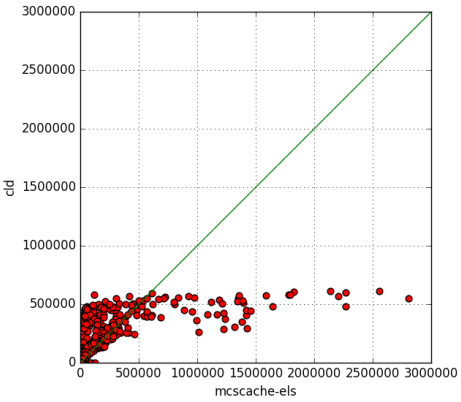

Fig. 2: mcscache-els vs mcsls-cld

the cache, without direct access to the original formula. On the other hand, querying the core database as currently implemented can still take a substantial amount of time on some instances. The query time seems to correlate with the average size of cores in the cache. In particular, on some instances cores can be very large (up to 200,000 clauses), which made the databases queries for the SAT solver time-consuming. In the present implementation, this seems to introduce an unnecessary overhead. This observation, together with the empirical results, motivates studying alternative ways of querying the database by taking into account the very simplistic form of the database, in order to to mitigate the observed negative effects. Alternatively, heuristics aiming at removing unused or too large cores could also be a viable and practical solution.

6 Conclusions

Analysis of over-constrained sets of constraints finds a widening range of practical applications. A central task in this context is the enumeration of minimal correction sets of constraints, namely, MCSes. Best-performing algorithms for the highly intractable task of MCS enumeration make high numbers of increasingly hard SAT solver calls as the number of MCSes increases. Motivated by this, we developed caching mechanisms to speed-up MCS enumeration. By keeping a global database in which unsatisfiable cores found during the computation of MCSes are stored, future calls to the SAT solver in the computation of new MCSes can be substituted by a polynomial-time check. In particular, the global database can be represented with a monotone formula, and even queried with low overhead using an off-the-shelf SAT solver. Empirical results confirm that caching is effective in practice, bringing significant performance gains to a state-of-the-art MCS algorithm. These results encourage further research on the topic. The development of dedicated solvers for handling the database represents a promising line of future research. Also, research on forgetting heuristics to keep the the database small could improve performance. Finally, future efforts will target the integration of caching within different MCS extraction algorithms.

References

1. M. F. Arif, C. Mencía, and J. Marques-Silva. Efficient MUS enumeration of Horn formulae with applications to axiom pinpointing. In *Proc. SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2015.
2. F. Bacchus, S. Dalmao, and T. Pitassi. Solving #SAT and Bayesian inference with backtracking search. *Journal of Artificial Intelligence Research*, 34:391–442, 2009.
3. F. Bacchus, J. Davies, M. Tsimpoukelli, and G. Katsirelos. Relaxation search: A simple way of managing optional clauses. In *Proc. AAAI*, pages 835–841. AAAI Press, 2014.
4. J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proc. PADL*, volume 3350 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2005.
5. P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Formula caching in DPLL. *Transactions on Computation Theory*, 1(3):9:1–9:33, 2010.
6. E. Birnbaum and E. L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*, 15(1):25–46, 2003.
7. Z. Chen and S. Toda. The complexity of selecting maximal solutions. *Information and Computation*, 119(2):231–239, 1995.
8. S. A. Cook. The complexity of theorem-proving procedures. In *Proc. STOC*, pages 151–158. ACM, 1971.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.
10. A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 26(1):53–62, 2012.
11. É. Grégoire, J. Lagniez, and B. Mazure. An experimentally efficient method for (MSS, coMSS) partitioning. In *Proc. AAAI*, pages 2666–2673. AAAI Press, 2014.
12. M. Kitching and F. Bacchus. Symmetric component caching. In *Proc. IJCAI*, pages 118–124, 2007.
13. T. Kopp, P. Singla, and H. A. Kautz. Toward caching symmetrical subtheories for weighted model counting. In *Proc. AAAI Beyond NP Workshop*, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016.
14. M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2016.
15. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.
16. Y. Malitsky, B. O'Sullivan, A. Previti, and J. Marques-Silva. Timeout-sensitive portfolio approach to enumerating minimal correction subsets for satisfiability problems. In *Proc. ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 1065–1066. IOS Press, 2014.
17. J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In *Proc. IJCAI*, pages 615–622. AAAI Press, 2013.
18. C. Mencía, A. Ignatiev, A. Previti, and J. Marques-Silva. MCS extraction with sublinear oracle queries. In *Proc. SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 342–360. Springer, 2016.
19. C. Mencía and J. Marques-Silva. Efficient relaxations of over-constrained CSPs. In *Proc. ICTAI*, pages 725–732. IEEE Computer Society, 2014.
20. C. Mencía, A. Previti, and J. Marques-Silva. Literal-based MCS extraction. In *Proc. IJCAI*, pages 1973–1979. AAAI Press, 2015.

21. P. Meseguer, N. Bouhmala, T. Bouzoubaa, M. Irgens, and M. Sánchez. Current approaches for solving over-constrained problems. *Constraints*, 8(1):9–39, 2003.
22. A. Morgado, M. H. Liffiton, and J. Marques-Silva. MaxSAT-based MCS enumeration. In *Proc. HVC 2012*, volume 7857 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2013.
23. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, pages 530–535. ACM, 2001.
24. A. Nöhrer, A. Biere, and A. Egyed. Managing SAT inconsistencies with HUMUS. In *Proc. VaMoS*, pages 83–91. ACM, 2012.
25. B. O'Sullivan, A. Papadopoulos, B. Faltings, and P. Pu. Representative explanations for over-constrained problems. In *Proc. AAAI*, pages 323–328. AAAI Press, 2007.
26. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
27. T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *SAT Online Proceedings*, 2004.
28. J. P. M. Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
29. J. Slaney. Set-theoretic duality: A fundamental feature of combinatorial optimisation. In *Proc. ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 843–848. IOS Press, 2014.
30. M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006.
31. C. Zielke and M. Kaufmann. A new approach to partial MUS enumeration. In *Proc. SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 387–404. Springer, 2015.